

MCP Server Architecture Patterns

Sonia Joshi

Sr. Solution Architect /Capgemini

Abstract—A practical architecture guide for designing secure, scalable and maintainable Model Context Protocol server ecosystems. This is complete guide prepared for enterprise architecture, development, security and platform teams.

I. INTRODUCTION

Model Context Protocol (MCP) is an open protocol that standardizes how AI applications connect to external data sources, tools, and services. It acts as a "universal connector" that enables AI assistants, agents, and applications to securely discover and interact with capabilities provided by external systems.

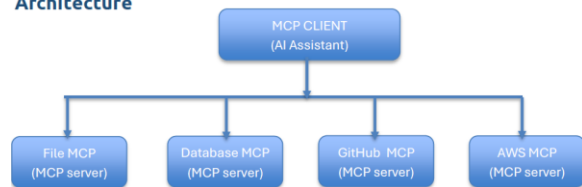
Model Context Protocol uses host-client-server architecture. A host manages one client connection per server, while servers expose focused capabilities through tools, resources and prompts. The patterns in this guide are architectural choices built on top of that protocol model, not mandatory MCP specification constructs.

Selecting the right Model Context Protocol (MCP) architecture pattern is critical because it directly impacts the scalability, security, maintainability, performance, and reliability of AI-enabled solutions. An architecture pattern determines how AI hosts interact with tools, services, data sources, and external systems.

II. SINGLE RESPONSIBILITY SERVER PATTERN

Each MCP server owns one well-defined domain or capability. This mirrors the single-responsibility principle and the MCP design goal of focused, composable servers. Examples include a Filesystem MCP, Database MCP, GitHub MCP, Jira MCP or AWS MCP. PROCEDURE FOR PAPER SUBMISSION

Architecture



Scenario: A source-control team operates a GitHub MCP independently from a database team. A repository outage does not affect customer-data lookup.

Typical use cases

- Focused integrations owned by separate teams.
- Capabilities with different security classifications.
- Services that require independent scaling or release cycles.

Apply these use cases when the capability has a clear business outcome, an accountable owner and measurable service expectations.

Key benefits

- Improved Maintainability: Each server contains a focused codebase and a well-defined responsibility, making updates, debugging, and enhancements significantly easier.
- Independent deployment and release cycle: Teams can deploy or upgrade individual MCP servers without impacting other integrations, reducing downtime and release risk.
- Clear credential and security boundaries. Access permissions can be isolated per domain. For example, a GitHub server only requires GitHub credentials, while an AWS server only requires cloud permissions, minimizing blast radius.
- Better scalability: High-demand services can be scaled independently. If database queries increase dramatically, only the Database MCP Server needs additional resources.
- Good fault isolation and testability. Smaller and focused servers enable more effective unit,

integration, and security testing with fewer dependencies.

- Technology flexibility: Different MCP servers can be implemented using the most suitable language, framework, or deployment model without affecting the overall ecosystem
- Fault isolation: A failure in one server (for example, GitHub API outage) does not impact unrelated capabilities such as filesystem operations or database access.

Drawbacks and risks

- More endpoints, deployments and operational objects.
- Tool discovery can become difficult at large scale.
- Cross-domain workflows require host or orchestrator coordination.

When to use

- A capability has a clear owner and lifecycle.
- Permissions differ significantly between domains.
- Failures must be isolated.

When not to use

Avoid this pattern when

- The capability is too small to justify a separate deployment.
- Several tools share the same business logic, data model, owner, and security requirements.
- Operating many small servers would create unnecessary infrastructure overhead.
- Users frequently require tightly coordinated operations across the same domain.

Implementation recommendations

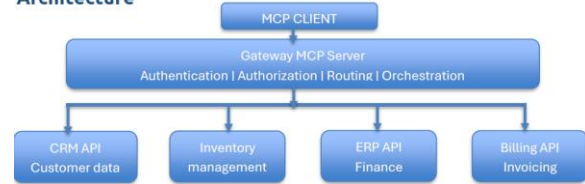
1. Define a precise server charter and ownership boundary.
2. Grant only the credentials required for that domain.
3. Keep tools cohesive; split the server if unrelated capabilities accumulate.
4. Publish version, owner, data classification and support information in a registry.

III. API GATEWAY MCP PATTERN

A gateway MCP provides a single governed entry point and routes or orchestrates calls to several backend APIs or domain services. It is useful where

AI clients should not know the details of every enterprise endpoint.

Architecture



Scenario: A customer-support assistant asks for customer status. The gateway combines CRM profile, ERP orders and billing balance into a single response.

Typical use cases

- Enterprise API estates. AI applications connect to one MCP endpoint instead of maintaining separate connections to every backend system. This simplifies Client configuration, Tool discovery, Connection management, Integration testing, Backend service onboarding.
- Centralized authentication, rate limiting and auditing. The gateway can enforce this restriction consistently, regardless of which backend application stores the information.

Key benefits

- Single endpoint for clients. This benefit improves enterprise adoption by reducing integration friction, clarifying accountability or limiting operational and security risk.
- Consistent security control: Security policies can be applied before requests reach internal systems. This reduces the risk of each backend service implementing controls differently.
- Backend System Abstraction: If one backend API changes, the gateway can adapt the integration without requiring every AI application to be modified.
- Reduced Tool and Integration Complexity: Exposing every backend API directly can create a large and confusing tool catalogue. The gateway can present a smaller set of business-oriented tools.
- Centralized Rate Limiting and Traffic Management: The gateway can protect backend systems from excessive or uncontrolled requests. Traffic-management capabilities may include: Per-user limits, Per-tool limits, Per-tenant quotas, Request throttling, Concurrency controls, Priority routing, Response caching

- These controls prevent an AI agent from unintentionally generating high request volumes that could affect the performance of critical enterprise systems
- Improved resilience: The gateway can implement standard resilience mechanisms for all downstream services. This can be achieved by setting: Timeouts, controlled retries, Circuit breakers, Fallback responses, Health checks, Load balancing, Cached responses, Partial-result handling
- Consistent Data Transformation: Different backend systems may use different formats, field names, identifiers, and protocols. The gateway can normalize this data before returning it to the AI application. This gives AI applications a predictable data model and reduces interpretation errors.
- Faster integration of Legacy systems: Legacy services can be connected behind the gateway through adapters without exposing their complexity to AI clients. Ggateway may integrate with SOAP services,
- Mainframe transactions, SAP interfaces, Oracle applications, Proprietary internal APIs, Older database procedures.

This allows organizations to introduce AI capabilities without immediately replacing stable legacy platforms.

Drawbacks and Risks

- Potential bottleneck and larger blast radius.
- Can become a monolithic super-server.
- Extra network hop and latency.
- Complex authorization if user identity is not propagated correctly.

When to use

- Many APIs require consistent governance.
- A curated business interface is preferable to raw APIs.
- Backend systems change independently of AI clients.

When not to use

Avoid this pattern when

- Only one backend system is involved.
- Direct MCP access is simpler and faster.
- Strong isolation is required between domains.
- The gateway could become a monolithic “mega server.”

- Centralization creates a single point of failure.
- Domain teams require independent deployments.
- Ultra-low latency is essential.
- The organization cannot support high availability, monitoring, and governance.

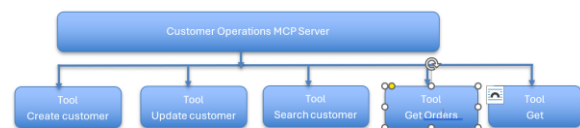
Implementation recommendations

- Keep domain business logic downstream; use the gateway for routing, policy and light composition.
- Deploy multiple instances with health checks, timeouts and circuit breakers.
- Propagate user identity and authorization context; never replace user authorization with a broad service account.
- Set per-tool rate limits and detailed audit records.

IV. TOOL AGGREGATOR PATTERN

A domain-oriented MCP server exposes multiple related tools through one cohesive interface. It balances the simplicity of one server with domain-driven organization.

Architecture



Scenario: Customer Operations MCP offers customer profile, order and invoice tools to support agents without exposing three separate systems.

Typical use cases

- HR operations
- Finance operations.
- Customer service.
- Supply-chain or procurement capabilities.
- Apply this use case when the capability has a clear business outcome, an accountable owner and measurable service expectations.

Key benefits

- Tools are easier for agents to discover. The Tool Aggregator Pattern organizes tools around a recognizable business domain rather than around technical systems or individual APIs. This approach makes the MCP ecosystem more intuitive because the organization of tools reflects how the business operates.

- Simplified tool discovery for agents: AI agents must identify the correct tool before completing a task. If tools are distributed across many small MCP servers, discovery and selection become more complicated.
- A domain aggregator presents a cohesive catalogue of related tools. For example, once the AI agent connects to a Customer Operations MCP Server, the agent can find all customer-related capabilities in one location. This can improve: Tool-selection accuracy, description consistency, Agent planning, Response reliability, Developer experience, Troubleshooting.
- Business-oriented naming and ownership. This benefit improves enterprise adoption by reducing integration friction, clarifying accountability or limiting operational and security risk.
- Fewer servers than one-tool-per-server. This benefit improves enterprise adoption by reducing integration friction, clarifying accountability or limiting operational and security risk.
- Consistent policies inside a domain. This benefit improves enterprise adoption by reducing integration friction, clarifying accountability or limiting operational and security risk.

Drawbacks and risks

- The server may become too large.
- Overlapping domains may cause ownership disputes.
- A large tool catalogue can reduce tool-selection accuracy.

When to use

- Tools share data, policies, vocabulary and ownership.
- A domain team releases the capabilities together.
- The tool count remains manageable and cohesive.

When not to use

Avoid this pattern when

- Tools belong to unrelated business domains.
- Tools require different security or compliance controls.
- Different teams own and release the capabilities independently.
- The server is becoming too large or difficult to maintain.

- Tools require significantly different scaling or availability levels.
- A server failure would affect too many business operations.

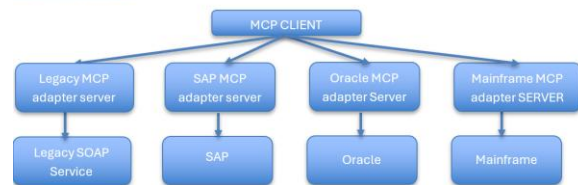
Implementation recommendations

- Group by business capability, not by coincidental technology.
- Use unambiguous tool names and descriptions.
- Review tool usage and retire duplicates.
- Split the server when security, scaling or ownership differs.

V. ADAPTER PATTERN

An MCP adapter wraps an existing system and translates MCP operations into legacy protocols, vendor APIs or proprietary interfaces. The backend remains unchanged.

Architecture



Scenario: An SAP adapter exposes `get_purchase_order` and `approve_purchase_order` while internally calling existing BAPIs and applying established controls.

Typical use cases

- SOAP and mainframe integration.
- SAP or Oracle access.
- Proprietary vendor platforms.
- Incremental modernization.
- Apply this use case when the capability has a clear business outcome, an accountable owner and measurable service expectations.

Key benefits

- Reuse Existing Enterprise Investments: Organizations often have critical business logic already implemented in legacy systems. Instead of rebuilding the complex processing system, using this pattern, adoption will be faster with low modernization cost, prevents technology investments.

- Instead of replacing these systems, the MCP adapter exposes their capabilities directly to AI applications.
- No invasive backend changes. This benefit improves enterprise adoption by reducing integration friction, clarifying accountability or limiting operational and security risk.
- Decouples AI systems from backend changes.

Drawbacks and risks

- Protocol translation adds latency and failure modes.
- Legacy limitations remain.
- Adapter must track backend schema changes.
- Risk of exposing unsafe low-level operations.

When to use

- Replacement of the backend is not practical.
- A stable legacy capability must be AI-accessible.
- Modernization must be incremental and low risk.

When not to use

Avoid this pattern when

- The backend already supports MCP or modern APIs directly.
- Protocol translation adds unacceptable latency.
- The legacy system is unstable or scheduled for replacement.
- Extensive new business logic is required in the adapter.
- Backend limitations prevent secure or reliable integration.
- Maintaining the adapter would cost more than modernizing the system.

Implementation recommendations

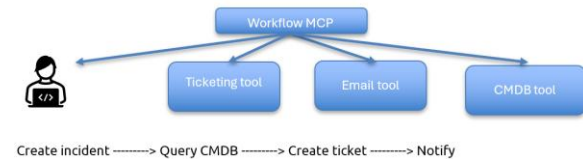
- Keep the adapter thin and place business rules in the source system.
- Translate backend errors into safe, actionable MCP errors.
- Add input validation, idempotency and timeouts.
- Expose business-safe actions rather than generic execute-command tools.

VI. ORCHASTRATOR/AGENT PATTERN

The Orchestrator (Agent) MCP Pattern introduces a centralized decision-making layer that coordinates multiple MCP servers, tools, APIs, and workflows to accomplish complex business tasks. Rather than a user

or AI application directly calling individual MCP tools, an intelligent orchestrator determines which tools to invoke, in what sequence, and how to combine the results.

Architecture



Scenario: An incident workflow queries the CMDB, opens a ServiceNow ticket, assigns the correct resolver group and sends a notification.

Typical use cases

- IT service management.
- DevOps remediation.
- Employee onboarding.
- Order fulfilment and approvals.
- Apply this use case when the capability has a clear business outcome, an accountable owner and measurable service expectations.

Key benefits

- Encapsulates complex workflows.
- Intelligent decision making: Reduces decision burden on the model.
- Provides reusable and consistent processes. This benefit improves enterprise adoption by reducing integration friction, clarifying accountability or limiting operational and security risk.
- Centralizes business logic.

Drawbacks and risks

- High implementation and testing complexity.
- Partial failures can leave inconsistent state.
- Long-running workflows require durable state.
- Broad privileges can increase risk.

When to use

- The process spans multiple systems.
- Steps must occur in a controlled order.
- Compensation, approval or human checkpoints are required.

When not to use

Avoid this pattern when:

- Simple single-tool operations

- Small standalone MCP servers
- Low-latency use cases requiring direct access
- Environments where orchestration overhead outweighs benefits

Implementation recommendations

- Model workflows explicitly with correlation IDs and durable state.
- Make actions idempotent and define compensation for partial failure.
- Require approval for sensitive or irreversible steps.
- Keep domain operations in downstream servers; the orchestrator coordinates rather than duplicates them.

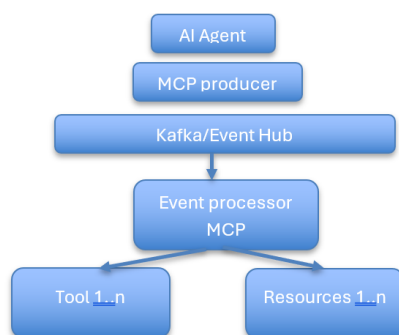
VII. EVENT DRIVEN MCP PATTERN

An event-driven implementation connects MCP capabilities to event streams or brokers. The event subsystem handles ingestion, while MCP tools allow users or agents to inspect, acknowledge or act on events.

The Event-Driven MCP Pattern enables MCP servers, agents, and enterprise systems to communicate through events rather than direct synchronous requests. Instead of one MCP server calling another and waiting for a response, systems publish events that interested consumers subscribe to and process asynchronously.

This pattern is widely used in enterprise architectures to build scalable, loosely coupled, and real-time AI ecosystems.

Architecture



Scenario: A SOC analyst uses `get_latest_alerts`, `investigate_alert` and `acknowledge_alert` while alerts continue to arrive through an event broker.

Typical use cases

- Monitoring and alerting.
- Security operations.

- IoT telemetry.
- Business-event investigation.
- IT Operations & AI Ops
- Apply this use case when the capability has a clear business outcome, an accountable owner and measurable service expectations.

Key benefits

- Near-real-time awareness.
- Loose coupling between producers and consumers.
- Scales for high event volumes.
- Enables proactive operational workflows.
- Enables autonomous AI agents - AI agents can subscribe to events and act independently.
- Improved fault tolerance -The event remains stored until processing resumes.
- Enhanced observability - Events create a traceable business activity stream.
- Supports Enterprise AI Ecosystems - One event can activate multiple AI agents.

Drawbacks and risks

- Duplicate, delayed or out-of-order events
- Backpressure and retention requirements
- Harder troubleshooting and replay
- Uncontrolled autonomous actions can amplify incidents

When to use

- Timeliness matters more than synchronous request-response
- Events already exist in Kafka, Event Hubs or another broker.
- Users need investigation and controlled response tools.

When not to use

Avoid this pattern when

- Simple request-response operations
- Low-latency transactions requiring immediate consistency
- Small standalone MCP deployments
- Use cases with very limited event volumes

Implementation recommendations

- Use domain events instead of technical events
- Keep ingestion separate from model interaction and store events durably.

- Use event IDs, deduplication and replay checkpoints.
- Summarize or filter high-volume streams before exposing them.
- Gate automated remediation with policy, confidence and approval thresholds.
- The Event-Driven MCP Pattern is one of the best patterns for building enterprise-scale AI and agentic architectures. It enables MCP servers and AI agents to react to business events in real time, supports loose coupling, and scales naturally across large organizations.

VIII. MULTI-TENANT MCP

The Multi-Tenant MCP Pattern is designed for scenarios where a single MCP server platform serves multiple customers, business units, departments, environments, or organizations (tenants) while ensuring complete isolation of data, security, configurations, and access controls.

Architecture



Scenario: A procurement SaaS MCP serves multiple companies, automatically scoping every supplier and purchase-order query to the authenticated tenant.

Typical use cases

- SaaS products.
- Enterprise shared platforms.
- Managed AI services.
- Regional or business-unit deployments.

Key benefits

- Lower infrastructure and operations cost.
- Scalable SaaS enablement
- Rapid & high scale customer onboarding
- Consistent upgrades and governance.
- Efficient resource utilization.
- Centralized security governance, observability and management.

Drawbacks and risks

- Tenant-data leakage is a critical risk.
- Noisy-neighbour performance issues.

- Complex tenant-specific policy and configuration.
- Compliance may require dedicated deployment.

When to use

- Many tenants use the same capability model.
- Strong logical isolation can meet requirements.
- Centralized lifecycle management is valuable.
- This is used with tenant context enforcement, data segregation, high security policy enforcement.

When not to use

Avoid this pattern when

- Highly regulated workloads requiring complete physical isolation
- Small deployments with one customer
- Highly customized customer-specific platforms
- Ultra-sensitive environments where dedicated infrastructure is mandatory

Implementation recommendations

- Resolve tenant identity from trusted authentication, never from model-supplied text alone.
- Apply tenant filters at every data boundary and use tenant-scoped credentials where possible.
- Test isolation with automated negative tests.
- Offer dedicated instances for tenants with exceptional regulatory requirements.

IX. MICROSERVICE MCP PATTERNS

Each backend microservice, or each major bounded context, exposes a corresponding MCP interface. Ownership and deployment align with the existing microservice organization.

The Microservice MCP Pattern applies microservices architecture principles to MCP servers. Instead of building a large monolithic MCP server that contains all business capabilities, functionality is split into multiple independent MCP services, where each service owns a specific business domain.

Each MCP service can be developed, deployed, scaled, monitored, and maintained independently. Core Principle is One Business Domain = One MCP Service.

Architecture

Scenario: Orders, payments and shipping teams independently publish business-safe MCP tools while a gateway presents a curated catalogue to AI hosts.

Typical use cases

- Cloud-native product platforms.
- Organizations with mature domain teams- digital, banking, healthcare domains
- Services require independent scaling.
- Rapidly evolving business capabilities.

Key benefits

- Independent release cycles, faster deployment.
- Clear team and domain ownership.
- Support Strong enterprise scale and fault isolation.
- MCP interface evolves close to the service.

Drawbacks and risks

- Server and tool proliferation.
- Discovery, versioning and policy consistency become difficult.
- Chatty cross-service workflows.
- Operational overhead.

When to use

- A mature microservice platform already exists.
- Teams own services end to end.
- Gateway, registry and observability capabilities are available.
- In the continuous delivery environment.

When not to use

- Avoid this pattern when
- Small projects
- Single-team applications
- Limited business functionality
- Simple proof-of-concepts
- Organizations without DevOps maturity

Implementation recommendations

- Do not automatically create one MCP server per tiny technical service; align with meaningful business domains.
- Use a registry and gateway for discovery and governance.
- Standardize naming, errors, telemetry and version policy.
- Prefer coarse business operations over exposing every internal endpoint.

X. SECURE BROKER PATTERN

A trusted host or broker mediates access between the AI experience and MCP servers. It filters context, evaluates policy, obtains consent and passes only the minimum information and permissions required.

The Secure Broker MCP Pattern introduces a centralized security and governance layer between AI applications and MCP services. Instead of AI agents, copilots, or applications connecting directly to MCP servers, all requests pass through a Secure Broker, which enforces authentication, authorization, security policies, auditing, compliance controls, and traffic management.

This pattern is particularly important in enterprises where MCP servers expose sensitive business systems, customer data, financial information, or regulated workloads.

Scenario: A finance assistant requests invoice lookup. The broker passes customer ID and approved scope, not the full conversation, and blocks payment changes without explicit approval.

Typical use cases

- Regulated enterprises.
- Sensitive customer or employee data.
- Cross-domain AI assistants.
- High-risk tools with side effects.

Key benefits

- Minimizes context and credential exposure.
- Strong Authentication and fine grain authorization.
- Centralizes consent and policy.
- Improves auditability and compliance.
- Threat detection and monitoring
- Preserves network isolation.

Drawbacks and risks

- Potential single point of failure
- Adds policy and integration complexity.
- Can create latency or availability dependency.
- Poorly designed central policy may block legitimate work.
- Broker compromise has broad impact.

When to use

- Data is sensitive or regulated.
- Several servers have different trust levels.
- User approval is required for consequential actions.
- AI agents with autonomous actions
- External customer access
- Zero trust architectures

When not to use

- Avoid this pattern when
- Small proof-of-concept environments
- Simple internal test systems
- Isolated single-user MCP deployments
- Scenarios where security requirements are minimal

Implementation recommendations

- Apply least privilege to data, tools and duration.
- Keep full conversation history in the host and send only task-relevant context.
- Mask secrets and sensitive fields before server invocation.
- Record user, tenant, tool, purpose, decision and result in tamper-resistant audit logs.

XI. LAYERED MCP ARCHITECTURE

A The Layered MCP Architecture Pattern organizes MCP servers and AI capabilities into distinct architectural layers, where each layer has a specific responsibility. This pattern follows traditional enterprise architecture principles and helps organizations build MCP ecosystems that are scalable, maintainable, secure, and governable.

Instead of creating a flat collection of MCP servers, responsibilities are separated into layers such as:

Experience Layer

Orchestration Layer

Business Services Layer

Integration Layer

Data Layer

This creates clear boundaries and reduces architectural complexity.

Scenario: A Finance MCP exposes invoice tools at the interface layer, applies approval and masking rules in the business layer, and calls ERP APIs through an integration layer.

Typical use cases

- Production-grade MCP servers.
- Systems requiring automated testing.
- Multiple backend integrations.
- Long-lived enterprise products.

Key benefits

- Clear separation of concerns.
- High testability and maintainability.
- Replaceable integrations and transports.
- Consistent validation and policy.
- Security by design

Drawbacks and risks

- More files and abstractions.
- Risk of unnecessary ceremonies for small servers.
- Poor boundaries can create pass-through layers.

When to use

- The server contains non-trivial policy or integration logic
- Several tools reuse the same services.
- The product will evolve over time.

When not to use

Avoid this pattern when

- Small proof-of-concept projects
- Single MCP server deployments
- Simple integrations with limited functionality
- Short-lived experimental solutions

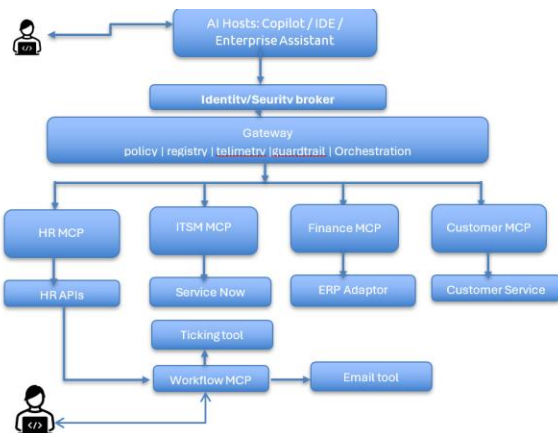
Implementation recommendations

- Keep lawyers independent
- Possible use of event driven communication
- Keep transport and JSON-RPC details out of business logic.
- Use typed requests and response models with validation.

- Centralize authorization, error mapping, logging and resilience concerns.
- Unit-test business logic and contract-test tools against real schemas.
- The Layered MCP Architecture is often considered the enterprise reference architecture for MCP ecosystems because it combines the strengths of multiple patterns.

XII. RECOMMENDED ENTERPRISE ARCHITECTURE

For most enterprises, the most effective and scalable MCP architecture is not a single pattern but a combination of complementary patterns working together. The recommended baseline consists of a Secure Broker, API Gateway, domain-oriented MCP services, and Adapter MCPs for enterprise applications. Orchestrators should be introduced selectively for business processes that genuinely require coordination across multiple domains or systems.



XIII. RECOMMENDED DESIGN PRINCIPLES

- Domain-based ownership and single responsibility
- Central authentication with end-user authorization propagation
- Least-privilege, short-lived credentials
- Human approval for sensitive or irreversible actions
- Structured audit logging and shared observability
- Stateless computing where possible, durable workflow state where necessary
- Explicit timeouts, retries, idempotency and circuit breakers
- Versioned schemas and compatibility testing

Decision Framework

Question	Guidance
Does one team own one cohesive capability?	Start with Single Responsibility or Tool Aggregator.
Are multiple-related tools exposed together?	Tool Aggregator MCP
Are many backend APIs hidden behind one governed interface?	Use API Gateway but retain downstream domain boundaries.
Is the backend legacy or proprietary?	Adapter MCP
Does a workflow span multiple systems and business steps?	Orchestrator MCP
Is input pushed continuously?	Use Event-Driven ingestion with controlled MCP actions.
Will one service support many customers?	Use Multi-Tenant controls or dedicated deployments.
Does the organization already operate microservices?	Align MCP interfaces with meaningful bounded contexts.
Is data sensitive or action consequential?	Place a Secure Broker and consent controls in the path.
Is the server production-grade?	Use a Layered internal architecture.
Is the platform a strategic enterprise solution?	Layered MCP Architecture

XIV. SECURITY AND OPERATIONAL CHECKLIST

- Every tool has an owner, purpose and data classification.
- Authentication identifies the user and tenant; authorization is evaluated per action.
- Servers receive only task-relevant context, never the full conversation by default.
- Tool inputs are validated, and outputs are filtered or masked.
- Writing actions are idempotent and protected by approval where appropriate.

- Secrets are stored in a vault, and credentials are short-lived.
- Timeouts, rate limits, retries and circuit breakers are configured.
- Audit logs include user, tenant, tool, parameters classification, decision, result and correlation ID.
- Metrics cover latency, errors, tool usage, denied requests and downstream health.
- Versions, deprecation timelines and compatibility tests are documented.

REFERENCES

- [1] Model Context Protocol, “Architecture overview,” Jul. 28, 2026. [Online]. Available:
- [2] Model Context Protocol, “Architecture,” *Model Context Protocol Specification*, Jul. 28, 2026. [Online]. Available:
- [3] Model Context Protocol, “MCP Python SDK documentation.” [Online]. Available: